FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Porting OpenThread to Resource Constrained Devices

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Stephan Felber
Registration Number 1426418

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dipl.-Ing. Stefan Seifried, BSc
Univ.Ass. Dipl.-Ing. Philipp Raich, BSc

Vienna, 1st April, 2019

Stephan Felber

Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Stephan Felber
Address

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. April 2019

Stephan Felber

Acknowledgements

My coffee machine without which I would not even have started university and my room mates without whom this thesis would have been finished a year earlier. Thanks boys.

Abstract

Small embedded systems featuring radio capabilities are getting cheaper and cheaper to produce. These devices are characterised by a comparably low processing power, a reduced periphery and low power consumption. To allow wireless communication between such devices, while respecting the restrictions, a new specially crafted communication protocol had to be designed. Demanding that it be power efficient, allowing devices to run on battery, error resistant and easy to use may seem like too much to ask, but Thread building on 802.15.4 and IPv6 fullfills all of the above.

This thesis presents an implementation of OpenThread, an implementation of the Thread specification for the Zolertia Z1 mote. The Z1 is a cheap and power efficient 16-bit micro controller.

By outlining the integral protocols of the Thread stack, the challenges of the thesis are presented. In the end, an implementation for OpenThread using the Zolertia Z1 is given. This implementation may be used to develop applications for OpenThread on a PC while taking part in an actual Thread network.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Aim of the Thesis 2

2 State of the Art **5**
2.1 802.15.4 6
2.2 Messages 8
2.3 6LoWPAN 8
2.4 Thread 10
2.5 Zolertia Z1 11

3 Methodology **13**
3.1 Tools 13
3.2 Settings 15
3.3 Evaluation 16

4 Implementation **19**
4.1 Direct Port 19
4.2 Host / Client oriented Port 21
4.3 Usage 25

5 Results **27**
5.1 Evaluation 27
5.2 Conclusion 31
5.3 Further Research 32

Acronyms **33**

Bibliography **35**

CHAPTER 1

Introduction

1.1 Motivation

As small computers, or so called embedded systems, got cheaper over the last few years, the need for specially tailored communication protocols arose. These embedded systems are restricted to a small power consumption and have to to be fairly cheap to produce, as they are expected to be used in large numbers. Designing a way of communication for such devices has to take these factors into account. It can neither be too complex to be run on a Micro controller Unit (MCU) nor too energy consuming to use.

The low cost per unit and the resulting volumes of units produced allow for local networks with a larger amount of participants. Connecting this local domain of communication to the Internet results in the so called Internet of Things (IoT). The IoT mainly consists of tiny computers which primarily do one or both of two things: firstly most of them serve as an actuator or a sensor. Like a light bulb that can be controlled via an app on your mobile phone, or a door lock that can be remotely released. Secondly they collect data. A humidity sensor might record the air quality, a smart dog feeder may remember when you feed your dog, and the light bulb might record when it is switched on and off. This data is then sent via the Internet to the user or the manufacturer.

An interesting sector for IoT is health care. As the median of the population age keeps getting older, new solutions for health care services have to be found. Because pretty much every bed is already connected to the Internet, it is expected that gathering all data available from a patient can be beneficial to both their recovery and further medical research.

As IoT devices are expected to get even cheaper and more numerous, more data sources will be monitoring the patients recovery story. The industry expects to develop more efficiently using such databases and keywords like Big Data, data mining and distributed intelligence are often associated with Health IoT [PZT+15].

1.1.1 Technical Motivation

Both the actuator and the sensor part of the connected device need a channel of communication to receive commands and deliver information [HL10]. Taking into account that the participants of the network should be allowed to spread about in the range of the network, a wireless communication is heavily favored because it eliminates the need of a cable tree to every node. Although wireless clears some difficulties, it also presents new challenges, like increased power consumption or interference on the shared medium.

There are nevertheless solutions to the challenges presented. One of them is Open Thread (OT) which builds on the 802.15.4 Physical Layer (PHY) standard. The 802.15.4 standard was crafted specifically with such low power devices in mind. In order to provide direct connectivity to the Internet, the IPv6 over Low Powered Personal Area Networks (6LoWPAN) layer translates Internet Protocol version 6 (IPv6) to 802.15.4 friendly packets. This promotes every participant in the network to a fully qualified member of the IPv6 Internet and grants all the benefits of a direct Internet access. OT is easing network setup and maintenance by providing self healing mechanisms and fully encrypted communication.

The Z1 features a 802.15.4 capable radio chip and multiple connection possibilities to connect sensors or actuators. It can run on battery or on the power provided by a Universal Serial Bus (USB) cable. Altough there are already multiple smaller devices providing the same functionality, the Z1 is still a candidate for the OT stack.

1.2 Problem Statement

The task of this thesis is to port the OT implementation to the Zolertia Z1 mote.

Up to now, OT is already running on multiple embedded platforms from different manufacturers, such as RIOT OS, Zephyr and provides two border router implementations. The porting process of the OT stack in general consists of writing a new Hardware Abstraction Layer (HAL). This means implementing an HAL between the program logic and the hardware it runs on.

OT is already a Thread certified implementation, which means that the Thread Group assures its compliance with the Thread standard. It was released under an open source license allowing developers easier development of applications for Thread.

The Z1 is in general not as powerful as the other hardware OT is already running on because it was released much earlier. This presents new challenges for the development of a HAL and problems like memory size or MCU capabilities may arise.

1.3 Aim of the Thesis

This thesis provides an implementation of the OT stack utilizing the Zolertia Z1 mote. As the Z1 provides a 802.15.4 compatible radio interface it allows the OT Stack to operate

either directly on the device or with some transport layer in between. This model is already similarly described on the official OT website as a network co-processor model, but no actual implementation was found at this time [OTs].

The result should allow developers writing applications for the OT stack to actively develop on a normal computer while still having access to actual radio hardware. Applications may be tested in an existing OT network and concurrently inspected by, for example, the GNU Debugger (GDB).

CHAPTER 2

State of the Art

This chapter provides a more exhaustive description of the standards and protocols that Thread relies on and are relevant for this thesis. Starting with the fundamentals, 802.15.4 will be explained, continuing with 6LoWPAN and its features. Thread, building on top of the previous two, is explained at the end. The corresponding Operating Systems Interconnection (OSI) model is visualized in figure 2.1.

Thread itself fullfills some aspects in the transport layer and some responsibilities of a network infrastructure manager. Considering this, it is both part of the transport layer and exceeds it. According to the OSI model, Thread corresponds to the transport layer, but Threads network management capabilities are simply not reflected in the OSI model. This correctly classifies Thread but neglects the efforts for the mesh network establishment and maintenance.

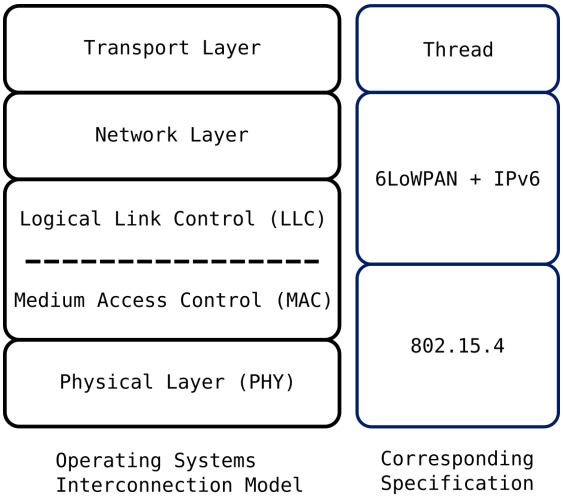


Figure 2.1: The first three layers of the OSI model on the left, and their respective specifications on the right.

2.1 802.15.4

In 2003, the Institute of Electrical and Electronics Engineers introduced the 802.15.4 standard to enable "very low-cost, low-power communication" for devices while ensuring consistent transmission and keeping the protocol simple but flexible. [80216, 13, 45]

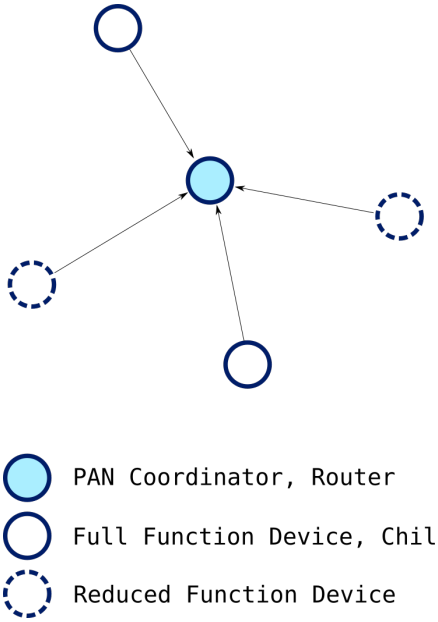
This standard proved to be the foundation for many communication stacks operating on power constrained devices. Since efficient energy saving communication requires fine-grained control over the hardware used, it was necessary to start such a communication stack at the very lowest OSI level, the PHY layer. This may be observed in figure 2.1.

The lowest layer in the OSI model states how the bits are delivered from one point to another. It takes care of the physical and mechanical aspects and ensures the actual delivery of the information.

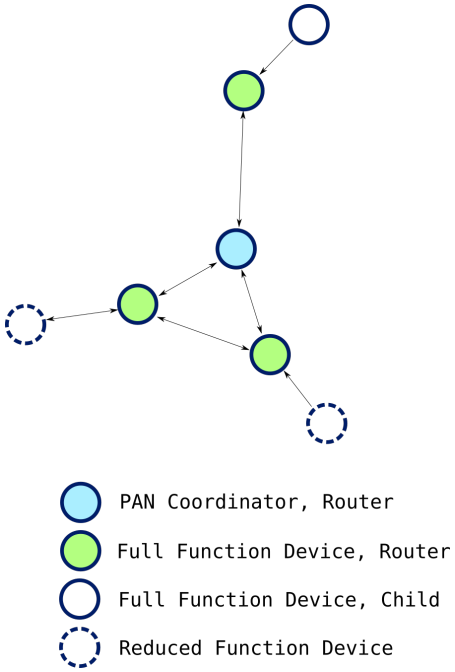
2.1.1 Topology

Two different network arrangements are allowed, called the star topology and the peer-to-peer topology. As the name suggests, the star topology forms a 'star' because every node connects to the same router, also called the Personal Area Network (PAN) coordinator. The single PAN coordinator handles all the message routing and every node participating in the network requires a connection to it, as visualized in figure 2.2a. This brings simplicity because it eliminates the need for routing algorithms, but also looses the ability to cover wider areas.

The peer-to-peer topology forms a mesh, which basically means that all routers connect to all other routers in range. A mesh is visualized in figure 2.2b. This implies that a router has to handle not only its children, but also maintains a table of its neighbouring



(a) The star topology as set in the 802.15.4 standard.



(b) The mesh, or peer-to-peer topology allows networks to cover a bigger area compared to the star topology. In the top right, there is a Full Function Device (FFD) that is not yet a router, because there is no need for it.

routers. This topology grants more redundancy to the network infrastructure but also brings more complexity. Message routing algorithms for finding the best path between two routers are required, but the network can span over a bigger area compared to the star topology. Thread uses the distance-vector routing protocol to find the ideal path between non adjacent routers [thr17, 112].

Both topologies deploy a single PAN coordinator which acts as the leader in infrastructure questions. It decides on new addresses, allows or denies new routers in a mesh network or sets the PAN prefix, to name only a few tasks.

2.1.2 Network Structure

Two different device types are specified, a FFD and a Reduced Function Device (RFD).

FFDs build the network. In the case of a star topology, one FFD assumes the role of the sole router and the PAN coordinator. It handles message routing, coordinates new nodes wanting to join the network and is also responsible for the connection to the Internet if available. If the sole router breaks, any other FFD that already joined the network can compete for the role as the new router and PAN coordinator.

The peer-to-peer topology, also called mesh, utilizes more than just one router [80216, 47]. Redundancy is achieved by simply deploying more than one device for the same task. If a router breaks, all of its connected children will try to reconnect to the network using another router in reach. If there was a FFD amongst the orphaned children, it will attempt to promote itself to a router, replacing the failed node.

If the network does not require another router, the FFD may simply work as an end node. It then waits until, for example, it receives a join request from a RFD, in that case it asks for a promotion to router status from the PAN coordinator and subsequently adds the new node as its child.

A RFD, as already mentioned, fully depends on FFDs for a working network infrastructure. In order to join a network a router or a FFD that is not yet a router needs to be in range. Since RFDs are not important to the network structure they can operate on much less power.

2.2 Messages

In an effort to ensure delivery over the generally unreliable nature of a wireless PHY layer, 802.15.4 specifies special acknowledge frames [80216, 105]. If the transmitting device requests a frame to be acknowledged by the receiving side, the receiver shall indicate through the so called Acknowledge (ACK) frame that the frame was correctly received. The standard sets a timeout within which the ACK has to be received, otherwise the frame is re-transmitted for a set amount of retries. If no ACK is received and all retries have been exhausted, the transmission of the frame fails.

Regarding security, messages may be encrypted using symmetric keys and can be authenticated using a cryptographic block cipher. [80216, 360] The standard does not set how the keys may be exchanged, but defines how messages are to be encrypted on the Medium Access Control (MAC) layer.

2.3 6LoWPAN

So far a structure for point-to-point communication was defined. Messages can be delivered to neighboring nodes, but up to now there is no concept of message routing to nodes with no direct line of communication. Considering that a connection to the Internet is a requirement, IPv6 seems like a good candidate for the network layer. Since IPv6 is expected to take a significant portion of all Internet traffic by 2019, [CAZ+14] and distributed low power networks typically exhibit a large amount of devices, IPv6 proved to be the better choice over Internet Protocol version 4 (IPv4). Its large address space allows for an individual address assigned to every node and eliminates the need for subnets.

The main problem 6LoWPAN solves, is that the Maximum Transmission Unit (MTU) of IPv6 is 1280 bytes while the 802.15.4 can only send 127 bytes at once. Using a

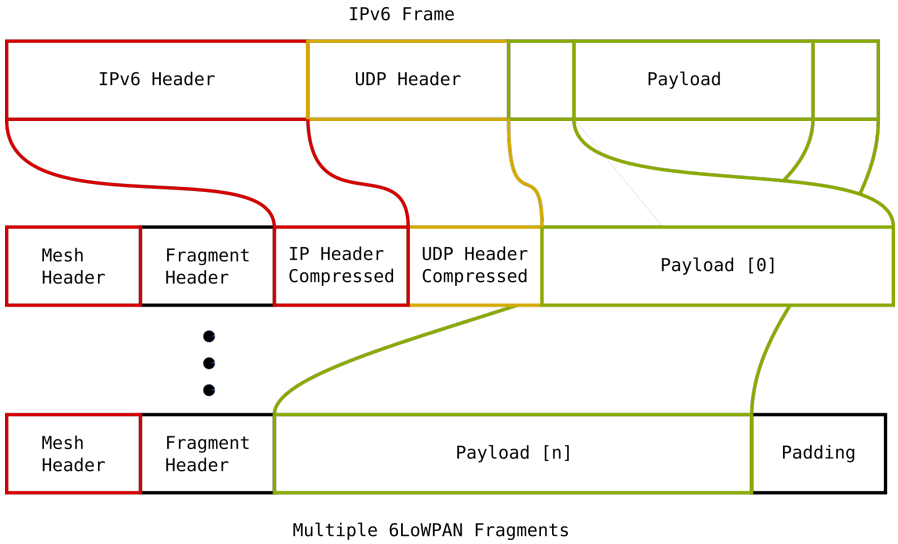


Figure 2.3: Boxes sharing the same border color also share the same information. The first bar shows the IPv6 frame, the second and the last bar model the first and the last fragment of the fragmented IPv6 packet.

conventional link, a full IPv6 packet could be sent on the PHY layer in one go, but on 802.15.4 it needs to be split up. This imposes longer latency for larger packets, which is justified by the assumption that 6LoWPAN networks are expected to rather send small packets periodically anyway [MKHC07, 5]. This nevertheless calls for an adaption layer between 802.15.4 and IPv6, 6LoWPAN to take care of efficient packet fragmentation, packet routing and compression. Since 802.15.4 reaches up to the OSI model layer 2 and IPv6 operates on layer 3, 6LoWPAN bridges the gap in the middle of those two.

2.3.1 Fragmentation and Reassembly

If an IPv6 packet is too big to be sent in one go, it will be split into multiple fragments that are sent independently. On the receiver side, the fragments are reassembled and presented as one packet to the next layer. This is achieved by preceding the packet with a fragmentation header which contains the index of the fragment and the total size of the packet. Allowing the receiver to efficiently assemble the whole packet even if the fragments arrive out of order [MKHC07, p. 11]. In contrast to IPv4, the MTU in 802.15.4 networks is guaranteed to be 127 bytes on every link which allows every router to forward a fragment without fully assembling it. Figure 2.3 shows a fragmented 6LoWPAN packet with a User Datagram Protocol (UDP) payload.

2.3.2 Header compression

Some assumptions about the PAN allow 6LoWPAN to strip the IPv6 header down to just the necessary information. In the best case, assuming a link local communication, the 40

bytes IPv6 header can be compressed to just 2 bytes. If the packet has to pass multiple routers on its way to the destination, a special mesh header is required. In this case, the total 6LoWPAN header can be as small as 7 bytes. Providing further compression, the following UDP, Transmission Control Protocol (TCP) or Internet Control Message Protocol (ICMP) headers may also be reduced in size. [MKHC07, p. 19]

If the message is larger than the MTU of 802.15.4, the 6LoWPAN fragmentation header introduces only 3 bytes resulting in a 5 byte overhead. A small packet travelling just in the mesh network consists of a mesh header and the compressed IPv6 header, adding up to only 6 bytes. [Mul07]

2.3.3 Link Layer Packet forwarding

As the mesh header of a packet is transmitted first, intermediate routers may need to forward the packet to the next stop at the link layer. This saves time as the upper layers do not have to be consulted. In figure 2.3, it can be seen that the mesh header also precedes fragmented packets and thus allows link layer forwarding.

2.4 Thread

With 6LoWPAN providing means of addressing nodes in and outside of a network, only a protocol for building and maintaining the network is missing. Thread fills this gap and provides features like self healing, nearly automated network joining, full data encryption and UDP plus optionally TCP as a transport layer.

2.4.1 Devices

In addition to the FFD and the RFD, Thread utilizes three more devices types. A Network Leader, a Router Eligible End Device (REED) and a Border Router. The RFD is simply renamed to a Sleepy End Device (SED). Before a network becomes usable, all the routers participating will elect a Network Leader among them. The leader takes decisions for the whole network, such as which address a new node receives or if a REED gets promoted to a router. A typical Thread network imposes a limit of 32 routers. A REED can ask the Network Leader to be promoted if another node tries to join the network using that REED as a parent. In the same manner, if all children of a router have disconnected, it will be downgraded to a REED again.

The Border Router typically utilizes different hardware compared to the other Thread devices, as it is also connected to some other network. In most cases, this works over Wi-Fi or Ethernet. A Border Router also routes messages within the mesh network as it still is a complete router. Thread allows for more than one Border Router.

2.4.2 Network Structure

Children of a router can be either REEDs or SEDs The latter received their name because they are allowed to sleep for short time frames to save energy and typically run on battery.

Thread networks form a mesh, as illustrated in figure 2.2b, which grants redundancy for certain nodes. A network consisting of just one router and multiple SEDs simply turns into a star topology. If the router representing the parent of a SED breaks, the SED can and will connect to another router or REED in its range. This happens automatically without the need for user interaction. Even if the acting Network Leader breaks, the other routers will elect a new one among them.

To form the network in the first place, Thread employs special messages called Mesh Link Establishment (MLE) messages. These are used to discover neighbors, test link quality and distribute route configuration. The link quality in Thread is not necessarily reflexive. This means that in a 802.15.4 network the link quality from device A to B may differ from the link quality from B to A. As a result, Thread can choose to route messages from B to A over an intermediate router and ensure safe delivery. MLE messages are not just used during the setup, but periodically utilized allowing the network to adjust dynamically.

2.4.3 Transport Layer

To actually deliver messages, every Thread device has to provide a UDP implementation, the TCP implementation is optional.

Starting with the authentication process, all communication in the network is encrypted. Every device "must be specifically authenticated and authorized" to join the network and complete a Transport Layer Security (TLS) handshake to agree on a common key [thr17]. All further communication is then secured with a network key.

2.5 Zolertia Z1

The Zolertia Z1 mote data sheet was released in March 2010 and predates the initial release of Thread by 5 years. The equipped hardware on the board also dates back to the earlier stages of embedded systems with a wireless connection. Its most important feature is the CC2420 chip from Texas Instruments which brings full 802.15.4-compliant radio communication. The Z1 allows up to 92 KB of program logic to be stored in flash memory, and provides 8 kB of RAM. The data sheet states that the chip brings 16 MB of flash, but the linker script provided by the manufacturer only provided 92 KB. As no further time was spent in investigating this discrepancy, the 92 KB of memory are assumed. The Z1 features a TI MSP430 16 bit MCU with up to 16 MHz clock speed.

In comparison, the TI CC2538 32 bit MCU, a device OT already supports, features 32 kB of RAM, 512 kB of flash memory and a Cortex M3. All further already ported

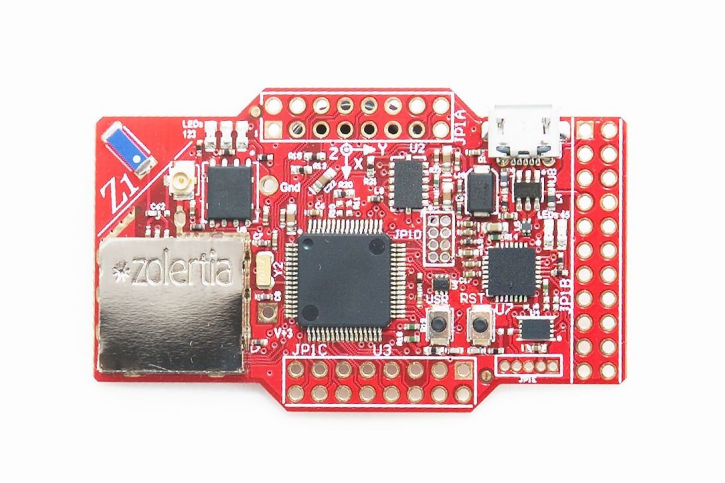


Figure 2.4: Zolertia Z1 mote, the silver plate on the right covers the radio chip. Image taken from [Zol10]

platforms also feature a 32 bit MCU which hints the first complications concerning the 16 bit Z1 platform.

CHAPTER 3

Methodology

Projects such as the OT stack utilize a HAL, which allows a developer to assume a reasonable hardware interface and write the application against it. To actually build the executable, a specific HAL is chosen that maps the abstract interface to an actual hardware.

Obvious advantages are less code duplication and the core logic of the application is less exposed to errors as it may be tested independently. It is also easier to distribute the same application to a variety of hardware without any modifications. For example, Google utilizes a HAL to distribute Android to different cellphones.

The HAL of the OT stack includes access to a 802.15.4 capable radio chip, non-volatile memory, hardware timers and various communication interfaces.

3.1 Tools

This section lists all the tools used and developed over the course of this thesis.

3.1.1 Linker Map Analyzer

In order to analyze the memory issues encountered during the development of the solution, a Linker Map Analyzer was developed [1]. Using Python, it parses all information present in the map file and stores it in a two dimensional linked list. It is then possible to run queries over the sections of the resulting hex file.

For example, just a specific section may be printed, all sections in `.text` sorted by size, or just one function. This example is visualised in listing 3.1. The output of the linker map analyzer allowed for a estimate of how many functions need to be removed in order to fit the implementation into the provided flash.

[1] `https://notabug.org/agentcoffee/stats`

`stats.py -f out.map [-p [-s text] [-i otMle] [-o]] [-t]`	
`-p`	Prints all the entries in the linker map matching the selection criteria.
`-s <section>`	Selects just entries in the specified section.
`-i <object>`	Selects just entries that, in a C++ manner, are instantiated from object. This allows for example, listing all functions in the object Mle (Mesh Link Establishment).
`-o`	Orders (sorts) all selected entries by size.
`-t`	Prints a table of all sections with their size at the very end.

Table 3.1: The usage of the linker map analyzer explained in more detail.

```
...
0x380   text       ot::Lowpan::Lowpan::Compress(...)
0x402   text       _printf_i
0x406   text       ot::Mle::Mle::HandleParentResponse(...)
0x428   text       addSetting.isra.1
0x43a   text       ot::Cli::Interpreter::ProcessCounters(...)
0x48c   text       ot::Mle::Mle::HandleUdpReceive(...)
0x4a2   text       ot::Lowpan::Lowpan::DecompressBaseHeader(...)
0x4e0   text       mbedtls_sha256_process
text: 0x18356(99158), listed: 0x18356(99158)

Section table:
...
0x1602  bss
0x161b  rodata
0x18356 text
```

Listing 3.1: A sample output of the script with the arguments `-f ot-cli-mtd.map -p -o -t -s text`. The number behind `text:` states the current size of the `.text` section which itself is bigger than the available flash memory in the Z1. The `_printf_i` corresponds to the newlib implementation of the c library that is more memory efficient. It is also interesting that two 6LoWPAN functions are under the top eight biggest functions. The arguments taken by the functions have been removed in order to maintain readability.

Running on Python 3.7, the script reads one linker map file and analyzes it according to the arguments given. Obtaining the linker map file may be achieved by invoking the linker with the argument `-Map=linker_map_file`. Usage of the script is provided in table 3.1.

3.1.2 Flashing

The Z1 flash is written using a python script written by Chris Liechti and taken from the contiki project [2]. The script is also hosted at the implementation mirror.

3.2 Settings

This section lists all the settings necessary for the implementation.

3.2.1 Compiler Flags

To compile the project, the MSPGCC-TI version 5.01.02.00 was used.

The OT project used the following arguments to invoke the compiler.

```
COMMONCFLAGS  -DNDEBUG -minrt -mlare -mmcu=msp430f2617
              -mhwmult=16bit -I/usr/msp430-elf/include
              -L/usr/msp430-elf/lib/large -g -fdata-sections
              -ffunction-sections -Os
      CFLAGS  -Wall -Wextra -Wshadow -std=c99 $COMMONCFLAGS
    CXXFLAGS  -Wall -Wextra -Wshadow -std=gnu++11
              -Wno-c++14-compat $COMMONCFLAGS
     LDFLAGS  -Wl,-gc-sections -nostartfiles $COMMONCFLAGS
```

The Server/Client implementation then used the following arguments to run the compiler.

```
CFLAGS  -Os -g -mmcu=msp430f2617 -Wall -Wpedantic
        -fpack-struct -std=c99 -Iinclude/
        -I/opt/ti/mspgcc/include -Wl,-Map=out.map"
```

3.2.2 Universal Asynchronous Receiver / Transmitter (UART) settings

A Linux system was used to develop the software for the Z1 and control the device. The necessary configuration is set using `stty`, short for set teletype, as can be seen in listing 3.2. The `stty` program is part of the GNU Coreutils package [3].

```
stty -F /dev/ttyS0 115200 min 1 -parenb crtscts
```

Listing 3.2: The options to configure the serial device for usage on the host. The `/dev/ttyS0` corresponds to the actual serial device. The baud rate is set to 115200. The option `min 1` tells the serial driver to wait for at least one character before returning. Specifying `-parenb` disables the generation of a parity bit. The `crtscts` option enables flow control.

[2] https://github.com/contiki-os/contiki/tree/master/tools/zolertia

[3] https://git.savannah.gnu.org/gitweb/?p=coreutils.git;a=tree;f=src;h=938a6f26722ec723c52a9b88b490761b8f42c281;hb=HEAD

Figure 3.1: The test setup. The two devices are connected and powered via common micro USB cables.

3.2.3 OT version

The OT source code was initially cloned from the original git repository [4]. As changes had to be incorporated into the OT core itself, the whole repository is also hosted in a git repository [5] at the last commit for this thesis.

3.3 Evaluation

In accordance to the procedure given by OT in [ot:c] to validate a new HAL port, the implementation is evaluated through the following 4 steps.

1. Interaction with the command line interface. This point is fulfilled by executing a few commands, which is done during the next few points anyway.
2. Building a Thread network and joining it with another device. By powering one device alone, it detects that no other Thread network is already present and shall start a new one. Powering the second device, it should detect the already present

[4] https://github.com/openthread/openthread/tree/6805cfa81e181cf35770d0f24cb42cbb86a88594
[5] git@notabug.org:agentcoffee/openthread.git

Thread network and attempt to join. After successfully becoming a child node, the second device shall be promoted to a router.

3. Transmission speed. By sending seven ICMP echo requests from one device to the other, the transmission speed is measured. The Round Trip Time (RTT) is chosen as the method of evaluation. RTT measures the time a packet is traveling from point A to B and the time it takes for B to acknowledge the reception of the packet. In case of layer 3 in the OSI model, the acknowledge does not happen with an ACK frame, but with the same ICMP message.

4. Re-attaching after a reset. After joining the Thread network, the second device saves information about the currently joined Thread network in its non-volatile storage. After resetting the device, it shall automatically rejoin the same Thread network without user intervention.

CHAPTER 4

Implementation

In this chapter, two different approaches to achieve the goal stated are presented.

The direct port tries to run the OT stack directly on the Z1 by simply implementing a suitable HAL. This is the intended way of porting OT to another hardware. After encountering various obstacles as described in 4.1, a second approach to achieve is taken.

The host / client oriented approach involves rewriting the HAL of the Portable Operating Systems Interface (POSIX) implementation already provided by Nest to work with the CC2420 chip on the Z1. This means developing a protocol over a UART connection and a receiving logic on the Z1 that handles the communication with the CC2420, which is described in detail under section 4.2.

4.1 Direct Port

This section details the challenges and possible solutions encountered while trying to port the OT stack directly on to the Z1.

1. The first problem that appears are predefined enums. Since a lot of static values are hard coded in such enums, and not in `#defines`, the compiler tries to place them into `ints` that correspond to 16 bits on the Z1. Unfortunately, a lot of bit masks and values are bigger than 2^{16} which requires them to either be rewritten, or compiled with a C++ compiler, as C++ allows sized enums in its standard. This turns out to be a solvable challenge, although multiple files that are not part of the HAL have to be rewritten.

2. Furthermore, the flash memory of the Z1 is divided into two parts. This is due to the fact that the original instruction set architecture on the 16 bit processor was only able to address $2^{16} = 65536$ bytes of flash memory but the Z1 provides 93884

bytes. This led to the modification of the instruction set architecture to allow 20 bit pointers, which are able to span the whole flash memory. The interrupt routine could not follow the update, and so the interrupt vector has to sit at the top of the 2^{16} byte flash and thus divides the whole flash into two parts.

The relevant parts of the linker file may be seen in figure 4.1. This rather unusual setup is not fully supported by the compiler, and no real bin packing for sections is implemented. It is possible to place whole sections in either region, but this does not utilize the flash to its fullest potential. This implies that either the project is small enough to fit in either region, or it is sufficient that just the `.data` section goes in the upper flash region, which is supported by the compiler. It is definitely possible to assign each function section into either flash region, but this requires a hard coded linker file and is not really optimal for ongoing development.

3. The 8 KB of RAM are mostly consumed by two 5352 bytes large state objects, that hold the complete state of the OT stack. This leaves about 2840 bytes for the whole stack and heap. Further, all already supported devices feature at least 28 KB of RAM. To ensure correct functionality, memory checks to avoid stack and heap corruption would have been required.

4. To provide encryption, Thread includes the size-optimised `mbedtls`[1] library. As the CC2420 network chip on the Z1 provides a hardware implementation for the used Counter with CBC-MAC algorithm, this library was replaced by an interface to the CC2420 encryption capabilities.

5. As the OT stack uses the standard `printf` implementation to format strings, the much smaller newlib [2] was included which provides the size-optimised `iprintf` function which was used instead of the standard `printf`.

Over the course of developing the HAL for the direct port, it turns out that just the default example application already exceeds the available flash memory on the Z1, even with all the size optimistions mentioned above.

To visualize the issue of memory limitation, the linker map analyzer already introduced in section 3.1 comes in handy. As the biggest sections all correspond to functions that are essential to the correct operation of OT, which can be seen in listing 3.1, a greater amount of smaller functions would have to be removed. This is possible, but stripping the OT project in size while also keeping some core functionality alive, requires deeper knowledge of the OT implementation.

This led to the conclusion that porting the OT stack directly onto the Z1 presents challenges that exceed the workload assigned for this thesis. As a result to this conclusion, another approach to reaching the goal stated is taken.

[1] `https://tls.mbed.org/`
[2] `https://sourceware.org/newlib/`

```
MEMORY {
  ...
  RAM        : ORIGIN=0x1100,  LENGTH=0x2000 /* END=0x30FF, size 8192 */
  ...
  ROM (rx)   : ORIGIN=0x3100,  LENGTH=0xCEBE /* END=0xFFBD, size 52926 */
  HIROM (rx) : ORIGIN=0x10000, LENGTH=0x9FFF /* END=0x19FFF, size 40958 */
  ...
  VECT1      : ORIGIN=0xFFC0,  LENGTH=0x0002
  ...
  VECT31     : ORIGIN=0xFFFC,  LENGTH=0x0002
  RESETVEC   : ORIGIN=0xFFFE,  LENGTH=0x0002
}
```

Figure 4.1: The memory segment of the linker file for the TI MSP430. Non relevant parts have been replaced by dots. It can be observed that the Read only Memory (`ROM`), or Flash Memory, is cut by the interrupt vectors from 1 to 31 and `RESETVEC` and ultimately continues in section `HIROM`. The total size of the `ROM` and `HIROM` section is 93884 bytes.

4.2 Host / Client oriented Port

The result for this thesis consists of a small protocol on top of the UART connection that, in combination with the Z1, allows the 802.15.4 compatible radio to be controlled. The device runs a small server controlling the radio chip and providing access to received data. This setup is compareable to an external 802.15.4 compatible USB antenna. An existing OT HAL implementation was convinced to talk with the new antenna and is thus theoretically able to participate in an actual OT network.

The so called host runs all the OT logic and sends data to the device, in this case the Z1. In the concept of splitting the radio controller and the OT logic onto two different devices the radio controller, or device, is also called network co-processor. Thread mentions this concept but uses a different protocol to control the device. As no documentation or actual implementation can be found, the protocol described in the following section is developed [ot:a].

4.2.1 Protocol

Sending one command and a variable-size payload, the protocol can be seen as a memory interface. The command corresponds to the memory location, the data represents the data to be written there. As no extra information is transmitted, no additional features such as error correction or automatic retransmission are provided. Such extension may easily be added, but is out of the scope of this thesis.

The underlying UART transmits with a baudrate of 115200 baud, no parity, one stop

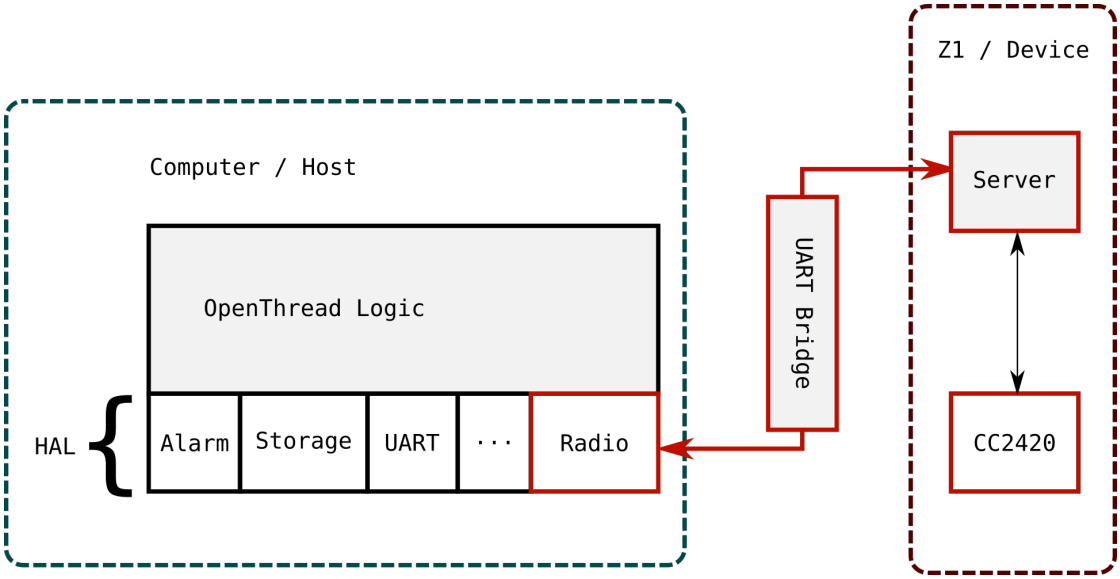


Figure 4.2: Implementation overview. All the parts in red have been implemented over the course of this thesis.

Bytes:	1	1	$0 \leq n \leq 255$
	Size (n)	Command	Data

Table 4.1: The frame structure of the UART protocol

and one start bit. A frame of the protocol contains the size of the data, the command and at most 255 bytes of data. The size information being sent first allows for easier assembly on the receiving part. As just one byte is reserved for commands or memory addresses, up to 256 commands could be implemented, although only a few are currently available. For a list of commands, see table 4.2.

4.2.2 Host

The host runs the logic of the OT stack. As the HAL implementation for POSIX-compliant platforms already exists, it can be simply adapted to forward the relevant data to the client. The reference implementation utilizes sockets to send data between different OT processes on the same system. This implementation enables the data to be sent over the air using the Z1 mote and allows the host to take part in an actual OT network.

This is achieved by cutting the OT stack in the radio implementation and rewriting the necessary functions to send information via the UART bridge to the device. Communication with the device is only done where necessary, if information that is already available on the host side allows the function to finish, then no communication is taking place.

Command	Description
OTPLATRADIOSETPANID	Sets the PANID on the CC2420, to enter the specified 802.15.4 network.
OTPLATRADIOSETEXTENDEDADDRESS	Sets a 802.15.4 extended address for the source address match feature. See OTPLATRADIOENABLESRC-MATCH and [80216, 103].
OTPLATRADIOSETSHORTADDRESS	Sets the 802.15.4 short address. See OTPLATRADIOENABLESRC-MATCH and [80216, 103]
OTPLATRADIOGETSTATE	Returns the debug state of the CC2420 as described in [cc213, 43].
OTPLATRADIOENABLE	Starts the CC2420 from its idle state.
OTPLATRADIODISABLE	Sets the CC2420 into its idle state.
OTPLATRADIOISENABLED	Returns true if the radio is enabled.
OTPLATRADIOSLEEP	Allows the CC2420 to sleep and save power.
OTPLATRADIORECEIVE	Sets the CC2420 explicitly into receive mode.
RADIOSETCHANNEL	
RADIOSETPSDULEN	Sets the length of the PSDU to be sent.
RADIOSETPSDU	Sets the actual PSDU.

Table 4.2: The commands implemented by the protocol.

The existing socket design already requires the whole PSDU to be constructed on the host which is then simply sent to the device. The payload of the PSDU is encrypted on the host using the `mbedtls` library.

As the OT stack requires an ACK (see section 2.2) to be received within the following 16 milliseconds after sending a frame, the ACK timeout has been raised to 40 milliseconds. In an attempt to minimise the overhead, the device already sends ACK frames itself and instantly transfers the message to the host. In theory, 16 milliseconds are enough for the acknowledge to reach the transmitter, but altering it to 40 milliseconds allows for a much stabler usage.

The host implementation with the corresponding HAL may be found at section 3.2.3.

4.2.3 Device

The device starts by initialising all the necessary components on the Z1 board. As the Z1 board features an adjustable oscillator, an iterative algorithm anneals the desired 8 MHz CPU frequency. Once the Serial Peripheral Interface (SPI) and the UART are running,

the CC2420 is initialised.

With the periphery set up, the device waits for a command to be received over the UART bridge. As soon as a full command is available, it is executed. In the regular case, this involves sending data to the CC2420. If the CC2420 receives a frame, it will alert the chip through an interrupt which schedules a read from the CC2420. The chip then copies the frame into an internal ring buffer and immediately transmits it to the host. The computer running the OT logic will buffer the data sent in the serial pipe and serve it once needed. This immediately clears the internal buffer of the CC2420, as it only has enough space for 127 bytes.

The receiving module on the device is structured into 3 different buffers as visualised in figure 4.3.

1. The first buffer is a standard ring buffer. The UART receive interrupt fills it with incoming bytes. If it detects that the next byte will overflow the ring buffer, the Clear To Send (CTL) line is asserted, indicating to the host that no more bytes may be sent. As soon as space is available again in the ring buffer, the CTL line is released again. This guarantees that no data is lost and the execution is not stalled for longer than necessary.
2. The second buffer slowly assembles a full command. The main loop periodically dispatches a routine that empties all available bytes from the ring buffer into the second buffer. As soon as the command is completed in the second buffer, it is copied into the third buffer and executed. This allows the next command to be assembled while executing. As the first byte in every telegram contains the length of the variable data section 4.1, it is used to detect when the whole command is received.
3. From the third buffer, the received instruction is executed. If said command requires data to be sent back, the information in question is handed to an asynchronous sending routine. This second interrupt driven routine transmits the data back to the host, where it is buffered in the serial file descriptor.

In theory, the device may stack up to 2 full commands plus the size of the ring buffer before the CTL line is set to logical 1. As most commands do not require the maximum data length available, more commands may be stacked in practice.

In an effort to reduce the overhead of the serial communication, a received message from the CC2420 is instantly transmitted to the host. The operating system will then buffer the message and deliver it to the OT stack once needed. This also reduces the overhead of reading the message on the host side, as the host does not have to wait for the serial communication and is just limited by the reading speed on the serial file descriptor.

If the message received by the CC2420 requires an ACK, the Z1 will instantly acknowledge the reception. The special acknowledge with data pending, required for routers with SED as children, is not implemented, yet.

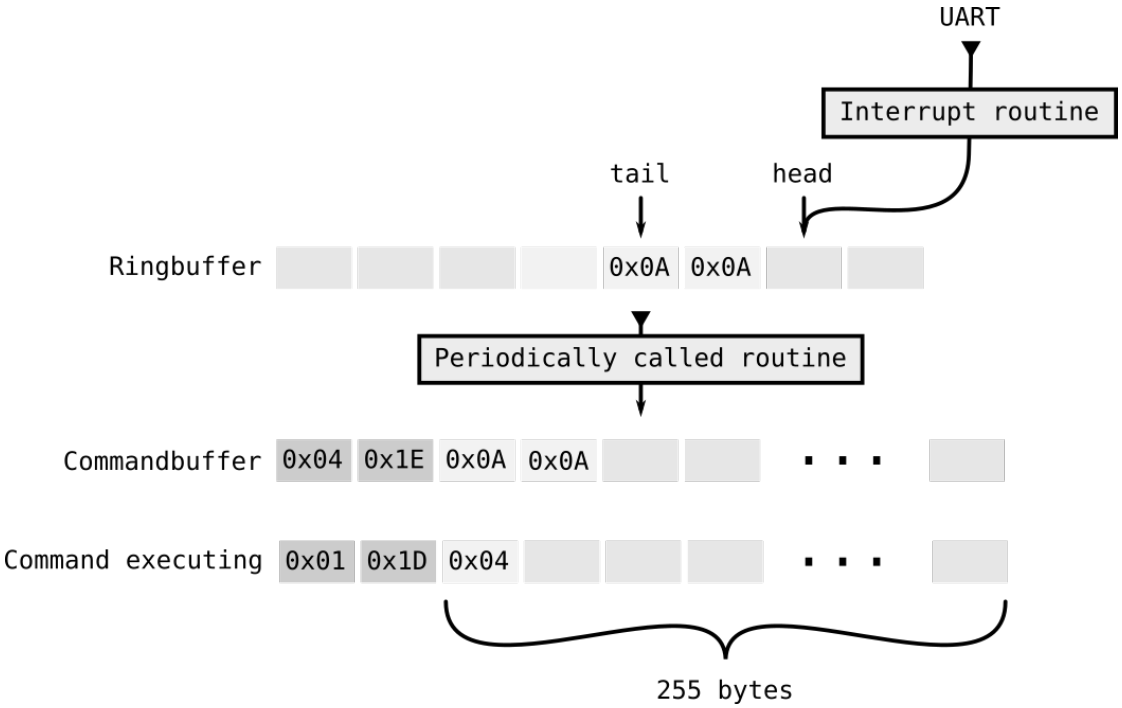


Figure 4.3: The 3 buffer system illustrated in detail. The ring buffer is filled by a UART interrupt routine and periodically copied into the command buffer. If the command is fully assembled in the command buffer, it is moved to the executing buffer and executed. In this example, the `RADIOSETPSDULEN (0x1D)` command is currently executed while the command `RADIOSETPSDU (0x1E)` is still being built up in the command buffer. As the name says, `RADIOSETPSDULEN (0x1D)` sets the length of the PSDU, which is also the length of the payload of `RADIOSETPSDU (0x1E)`. The actual PSDU was set to `0x0A` for illustration purposes.

The complete device implementation is available in a separate repository at [3], together with a flash script mentioned in section 3.2. It also contains a Makefile with targets to compile, flash and reset the Z1.

4.3 Usage

The original executable has been extended with a low level debug framework that simply prints everything received and sent through the UART bridge to a file provided through the `-d` argument. Passing it a pipe, previously created with `mkfifo`, allows observation of the log in real time as can be seen in figure 4.4. As the original OT framework sends all logs directly to the system log, the logging framework has been modified to write all

[3] `https://notabug.org/agentcoffee/z1_board`

`./ot-cli-ftd -f /dev/ttyUSB0 -n 2 -d dbg.fifo`	
`-f <device>`	The USB character device of the z1.
`-n <node id>`	The node ID of this specific node in the network.
`-d <debug pipe>`	A file object the protocol log should be written to. If it is not needed /dev/null has to be passed.

Table 4.3: The usage of the OT logic.

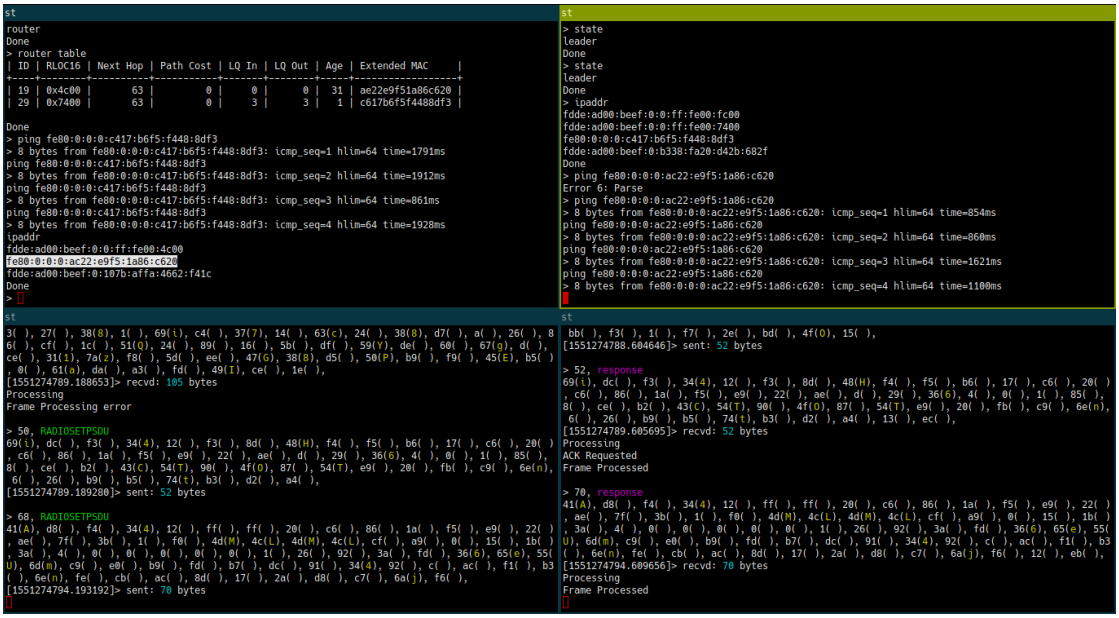


Figure 4.4: The typical developing setup. The two terminals at the top run the OT logic. The two terminals at the bottom show the debug output of the protocol implementation. Currently, two frames can be seen being sent from the left process to the right process via two Zolertia Z1 motes. The top two terminals also show the results of four pings.

logs to the file descriptor 3 that has to be piped to a file or just `/dev/null` if it is of no interest. OT will not work correctly if no pipe is passed.

After connecting and flashing the Z1, the corresponding character device has to be passed to the executable via the `-f` argument. Additionally, a node ID is required, using the `-n` parameter. The usage is specified in table 4.3

CHAPTER 5

Results

This thesis presents a functional implementation for the specified protocol together with a working HAL for OT. As the implementation is just expected to underline the feasibility of such an external network co-processor it may still miss features for a full OT compliance. For example, currently the special acknowledge frame with data pending flag is not implemented [80216, 152] as this feature is not needed for the proof of concept. Adding it is rather straight forward and no further pitfalls are expected.

5.1 Evaluation

The implementation was evaluated against the tutorial given in [ot:b] and defined in subsection 3.3. The test setup can be seen in figure 3.1

5.1.1 Interaction with the command line interface

As can be seen in figure 5.1, the command line works fine. The `help` command lists all available commands. Some of them may require additional arguments, a full reference may be found in the official OT repository [1].

5.1.2 Building a Thread network and joining it with another device

After powering the first device and letting it establish a Thread network, the second device is started. Follwing some negotiation, which usually takes abut 5 to 10 seconds, one device joins the other. The delay happens because the ACK may miss its time frame and thus invalidates the message, which crashes the current joining process. The reason was explained in further detail in section 4.2.2.

[1] `https://github.com/openthread/openthread/blob/master/src/cli/README.md`

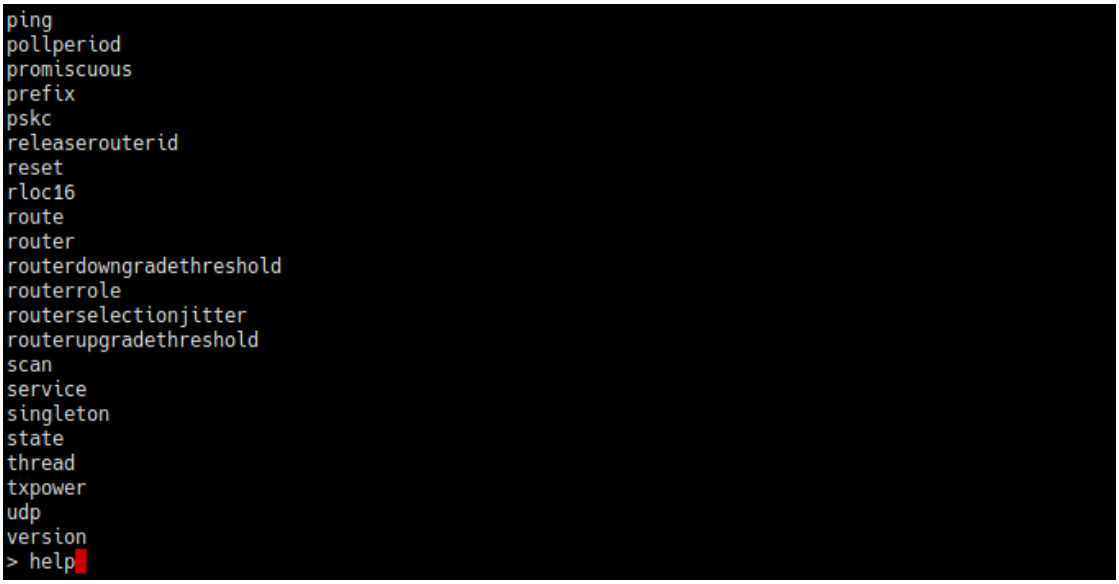


Figure 5.1: The tail of the `help` command can be seen. As the command line logic of OT was not modified, it is expected to work the same as before.

This may result in device A joining the Thread network of device B, even though A was started earlier. This happens because after a failed attempt to join a network the joining device assumes that the network has failed and starts its own. As soon as the other device notices the presence of the second network, it attempts to join. Figure 5.2 shows the joining process from the parents view. At first the `router table` contains only itself, because the child is not yet promoted to a router. After some time, the child is also promoted and shows up in the `router table`. One ICMP echo request has been sent and answered and the `router table` command shows that two devices are taking part in this Thread network.

In figure 5.3, the same process may be seen from the child side. After joining, the device automatically becomes a child and after some time it is promoted to a router. One ICMP echo request has been sent and answered. The `state` command tells us that the device has changed its state from child to a router and once the device has been promoted to a router, it also maintains a router table.

5.1.3 Transmission speed, RTT

The seven ICMP echo requests have been sent within 2 minutes and the devices were about 30 centimeters apart. Tables 5.1 and 5.2 list the results of the 14 pings.

The RTT from the parent to the child is on average faster here. That could imply that in general a router takes less time to answer than the network leader. This assumption requires more thorough testing but the fact that a network leader has more responsibilities than a router could explain the delayed ICMP response.

```
> state
leader
Done
> ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=1 hlim=64 time=1502ms
router table
| ID | RLOC16 | Next Hop | Path Cost | LQ In | LQ Out | Age | Extended MAC     |
+----+--------+----------+-----------+-------+--------+-----+------------------+
| 27 | 0x6c00 |       63 |         0 |     3 |      3 |  27 | c617b6f5f4488df3 |
| 37 | 0x9400 |       37 |         0 |     0 |      0 | 154 | ae22e9f51a86c620 |

Done
>
```

Figure 5.2: This figure shows the leader of the established Thread network as can be seen by the output of the `state` command while another device joins the Thread network.

```
> state
child
Done
> ping fe80:0:0:0:ac22:e9f5:1a86:c620
> 8 bytes from fe80:0:0:0:ac22:e9f5:1a86:c620: icmp_seq=1 hlim=64 time=1495ms
state
router
Done
> router table
| ID | RLOC16 | Next Hop | Path Cost | LQ In | LQ Out | Age | Extended MAC     |
+----+--------+----------+-----------+-------+--------+-----+------------------+
| 27 | 0x6c00 |       63 |         0 |     0 |      0 | 142 | c617b6f5f4488df3 |
| 37 | 0x9400 |       63 |         0 |     3 |      3 |  19 | ae22e9f51a86c620 |

Done
>
```

Figure 5.3: This figure shows the child joining an existing Thread network.

```
> ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=2 hlim=64 time=1115ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=3 hlim=64 time=871ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=4 hlim=64 time=1010ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=5 hlim=64 time=912ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=6 hlim=64 time=716ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=7 hlim=64 time=1204ms
ping fe80:0:0:0:c417:b6f5:f448:8df3
> 8 bytes from fe80:0:0:0:c417:b6f5:f448:8df3: icmp_seq=8 hlim=64 time=1146ms
```

Figure 5.4: The seven ICMP echo requests and responses from the network leader to its child.

Sequence	round trip time (ms)
1	1308
2	1272
3	1313
4	1298
5	1056
6	1216
7	1606
Mean	1295

Table 5.1: Seven ICMP echo requests from the child to the parent.

Sequence	round trip time (ms)
1	1115
2	871
3	1010
4	912
5	716
6	1204
7	1146
Mean	996

Table 5.2: Seven ICMP echo requests from the parent to the child as can be seen in figure 5.4.

5.1.4 Resetting a router and validate reattachment

Figure 5.5 shows the reset and re-attach procedure. After successfully joining the Thread network, the device is reset using the command `reset`. As the whole Thread stack is now in its original state, the whole interface has to be brought up again using `ifconfig up`. After restarting the Thread stack, the device immediately re-joins the network and assumes its previous role as a router. The `ping` at the very end confirms connectivity to the network.

One ICMP echo request is sent to the network leader to confirm actual connectivity with the network.

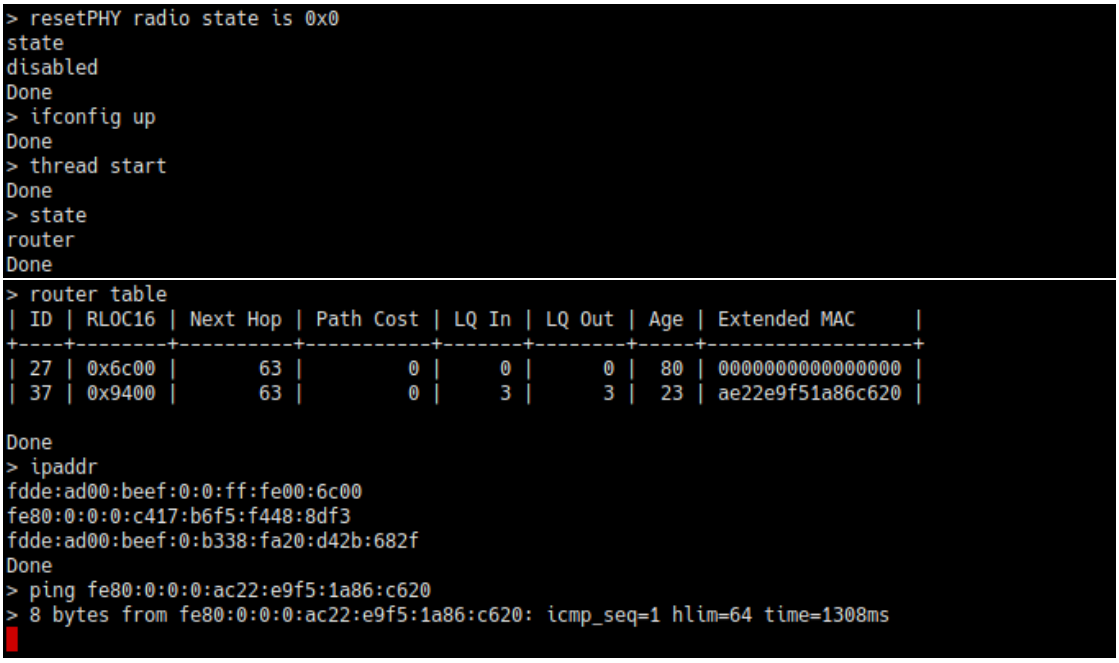


Figure 5.5: The reset and re attach procedure. Immediately after the reset, the debug message informing about the physical state of the radio is seen. There are two images as the total output had to be split over two pages of terminal output.

5.2 Conclusion

The results show a possible implementation of the so called network co-processor where the logic of the application runs on a different hardware than the network related logic.

This thesis also acknowledges that the timing constraints introduced by the UART connection may not be fixed at this stage of the OT stack. While certain optimisations may allow for a faster RTT or faster acknowledging of frames, the minimum time to transmit a full 802.15.4 frame to the device cannot be lower than 11 (ms) on a baud rate of 115200. Taking into account that it also takes another 11 (ms) for the receivig device to deliver the frame to the host, this introduces a delay of at least 22 (ms) simply due to transportation.

This may seem reasonable for a protocol as a whole, but as the delay is introduced at the HAL for a protocol stack that already has timing windows of 16 (ms) 4.2.2, it is not certain that the 22 (ms) do not violate other timing constraints. Through testing, this thesis assumes that other timing windows are met, but cannot guarantee it.

The new HAL works most of the time, but seeing that already one timing window in the stack had to be altered for stable operation, this is probably not the most efficient way to write a fully Thread certified HAL for OT.

Despite the solution still leaving room for improvement it presents an already working

state of OT utilizing the Z1. Of course it also allows applications to be developed against the OT stack and tested over a real wireless connection.

Testing it in an actual Thread network requires the special ACK with data pending to be implemented [80216, 152]. The mechanism the OT POSIX implementation uses to determine which devices receive a data pending flag can be mirrored to the device. All necessary commands may be added to the UART bridge.

5.3 Further Research

Implementing the missing ACK with data pending feature and testing the Z1 together with other Thread devices would be the next step building on top of this thesis. This would be the final step towards actual Thread certification for new HAL implementations [ot:c].

Assuming the timing restrictions imposed by the serial communication line are too high to deal with, moving more of the OT stack on to the device can speed things up. This is already proposed by the OT developers in form of a network co-processor, but no actual implementation was found at the time of writing [ot:a].

The Z1 is definitely a good candidate for a network co-processor implementation because of its power efficient operation, the 802.15.4 compatible radio chip and the rather fast UART connection to the host. It also features a rather large memory and flash, compared to devices from its time.

As the protocol developed through this thesis is not Thread specific, it can be adapted for any project taking part in 802.15.4 networks as required.

Acronyms

6LoWPAN IPv6 over Low Powered Personal Area Networks. 2, 8–10, 14

ACK Acknowledge. 8, 17, 23, 24, 27, 32

CTL Clear To Send. 24

FFD Full Function Device. 7, 8, 10

HAL Hardware Abstraction Layer. 2, 13, 16, 19–23, 27, 31, 32

ICMP Internet Control Message Protocol. 10, 17, 28–30

IoT Internet of Things. 1

IPv4 Internet Protocol version 4. 8, 9

IPv6 Internet Protocol version 6. 2, 8–10

MAC Medium Access Control. 8

MCU Micro controller Unit. 1, 2, 11, 12

MLE Mesh Link Establishment. 11

MTU Maximum Transmission Unit. 8–10

OSI Operating Systems Interconnection. 5, 6, 9, 17

OT Open Thread. 2, 3, 11, 13, 15, 16, 19–28, 31, 32

PAN Personal Area Network. 6–9

PHY Physical Layer. 2, 6, 8, 9

POSIX Portable Operating Systems Interface. 19, 22, 32

PSDU Physical Service Data Unit. 23, 25

REED Router Eligible End Device. 10, 11

RFD Reduced Function Device. 7, 8, 10

RTT Round Trip Time. 17, 28, 31

SED Sleepy End Device. 10, 11, 24

SPI Serial Peripheral Interface. 23

TCP Transmission Control Protocol. 10, 11

TLS Transport Layer Security. 11

UART Universal Asynchronous Receiver / Transmitter. 15, 19, 21–25, 31, 32

UDP User Datagram Protocol. 9–11

USB Universal Serial Bus. 2, 16, 21

Bibliography

[80216] Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.

[CAZ+14] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. Measuring ipv6 adoption. *SIGCOMM Comput. Commun. Rev.*, 44(4):87–98, August 2014.

[cc213] Cc2420 datasheet. *CC2420 Datasheet*, pages 1–85, February 2013.

[HL10] Y. Huang and G. Li. A semantic analysis for internet of things. In *2010 International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 336–339, May 2010.

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of ipv6 packets over ieee 802.15.4 networks. RFC 4944, RFC Editor, September 2007. `http://www.rfc-editor.org/rfc/rfc4944.txt`.

[Mul07] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, pages 78–82, New York, NY, USA, 2007. ACM.

[ot:a] Compile openthread for a [host / 802.15.4 controller] arrangement - stack overflow. `https://stackoverflow.com/questions/48672121/compile-openthread-for-a-host-802-15-4-controller-arrangement`. Accessed: 2019-03-04.

[ot:b] Simulating a thread network with openthread. `https://codelabs.developers.google.com/codelabs/openthread-simulation-posix/#3`. Accessed: 2019-02-27.

[ot:c] Validate the port | openthread. `https://openthread.io/guides/porting/validate-the-port`. Accessed: 2019-03-04.

[OTs] Platforms | Open Thread. `https://openthread.io/platforms`. Accessed: 27. 11. 2018.

[PZT+15] Zhibo Pang, Lirong Zheng, Junzhe Tian, Sharon Kao-Walter, Elena Dubrova, and Qiang Chen. Design of a terminal solution for integration of in-home health care devices and services towards the internet-of-things. *Enterprise Information Systems*, 9(1):86–116, 2015.

[thr17] Thread specification. *Thread Specification 1.1.1*, pages 1–348, February 2017.

[Zol10] Zolertia. *Z1 RevC Datasheet*, 3 2010.